numerics.hpp Documentation

Table of Contents

all definitions are members of namespace `numerics`, all examples assume:

```
using namespace std;
using namespace arma;
using namespace numerics;
using namespace ode;
```

Utility Methods

Modulus operation

```
int mod(int a, int b);
```

In `C++`, the modulo operator (`a%b`) returns the remainder from dividing `a` by `b`, while `mod(a,b)` function returns $a (\text{mod } b)$ which is distinct from `a%b` whenever `a` is negative.

Example:

```
int a = -3 % 11; // a = 2
int b = mod(-3, 11); // b = 8
```

Meshgrid

```
void meshgrid(arma::mat& xgrid, arma::mat& ygrid,
              const arma::vec& x, const arma::vec& y);
void meshgrid(arma::mat& xgrid, const arma::vec& x);
```

This function constructs a 2D grid of points (representing a region of the xy-plane), with non-uniform points defined in `x` and `y`.

Example:

```
vec x = linspace(0, 2, 10);
vec y = -cos(regspace(-M_PI, M_PI, 20))
mat XX, YY;
meshgrid(XX, YY, x, y);
```

Polynomial Derivatives and Integrals

Given a polynomial coefficient vector (produced from `arma::polyfit`), we can produce derivatives of the polynomial with the following function:

```
arma::vec polyder(const arma::vec& p, unsigned int k = 1);
```

where `p` is the coefficient vector, and `k` is order of the derivative. By default, `k = 1` which corresponds to the first derivative (and `k = 2` is the second derivative, and so on). The output is also a polynomial coefficient vector.

We can integrate the polynomial:

```
arma::vec polyint(const arma::vec& p, double c = 0);
```

where `p` is the coefficient vector, and `c` is the constant of integration; by default `c = 0`. The output is also a polynomial coefficient vector.

Sample Discrete Distributions

Given a discrete probability mass function, we can produce a random sample.

```
arma::uvec sample_from(int n, const arma::vec& pmf, const arma::uvec& labels=arma::uvec());

int sample_from(const arma::vec& pmf, const arma::uvec& labels=arma::uvec());
```

The output is either a set of integers between 0 and n-1 refering to the index associated with the pmf, or a set of labels sampled from `labels` using the pmf for random indexing.

Quadrature and Finite Differences

Integration

```
enum class integrator {SIMPSON,LOBATTO};

double integrate(const function<double(double)>& f,
                 double a, double b,
                 double err = 1e-5,
                 integrator i = LOBATTO);
```

We have multiple options for integrating a function $f : \mathbb{R} \to \mathbb{R}$ over a finite range. Primarily, we use `integrate()`. If f is smooth, then the default integrator is ideal, otherwise, we should opt to use Simpson's method.

```
double f(double x) return exp(-x*x); // e^(-x^2)
double lower_bound = 0;
double upper_bound = 3.14;

double I = integrate(f, lower_bound, upper_bound);

double I_simp = integrate(f, lower_bound, upper_bound, 1e-6,integrator::SIMPSON);
```

Given a very smooth function (analytic), we can approximate its integral with few points using polynomial interpolation. Traditionally, polynomial interpolation takes $\mathcal{O}(n^3)$ time, but since we can choose the set of points to interpolate over, we can use Chebyshev nodes and integrate the function in $\mathcal{O}(n \log n)$ time using the fast fourier transform (caveat, this implementation is not optimal, it is $\mathcal{O}(n^2)$ for now–still an improvement). The resulting error improves spectrally with n.

```
double chebyshev_integral(const function<double(double)>& f,
                          double a, double b,
                          unsigned int num_f_evals = 32);
```

Where `num_f_evals` is the number of unique function evaluations to use, which is 32 by default. Increasing `num_f_evals` improves the accuracy, but very few function evaluations are actually needed to achieve machine precision.

note: If the function is not (atleast) continuous, the approximation may quickly become ill conditioned.

Discrete Derivatives

Finite Differences

```
double deriv(const function<double(double)>& f, double x,
             double h = 1e-2, bool catch_zero = true);
```

This function uses 4th order finite differences with step size `h` to approximate the derivative at a point `x`. We can also ask the derivative function to catch zeros, i.e. round to zero whenever `f'(x) < h`; this option can improve the numerical stability of methods relying on results from `deriv()`, e.g. approximating sparse Hessians or Jacobians.

note: `deriv()` may never actually evaluate the function at the point of interest, which is ideal if the function is not well behaved there.

Example:

```
double f(double x) return sin(x);

double x = 0;

double df = deriv(f,x); // should return ~1.0

double g(double x) return cos(x);
double dg = deriv(g,x,1e-2,false); // should return ~0.0
/*
in this case we are better off if catch_zero = true because d(cos x)/dx = 0 for x = 0.
*/
```

We can also approximate gradients and Jacobian matrices:

```
arma::vec grad(const function<double(const arma::vec&)> f,
               const arma::vec& x,
               double h = 1e-2,
               bool catch_zeros = true);

arma::mat approx_jacobian(const function<arma::vec(const arma::vec&)> f,
                     const arma::vec& x,
                     double h = 1e-2, bool catch_zero = true);

arma::vec jacobian_diag(const function<arma::vec(const arma::vec&)> f,
                        const arma::vec& x,
                        double h = 1e-2);
```

These functions are wrappers for the `deriv()` function. So the functionality is similar. The function `jacobian_diag()` computes only the diagonal of the jacobian matrix, which may only make sense when the jacobian is $n \times n$.

Example:

```
double f(const vec& x) return dot(x,x); // return x^2 + y^2 + ...

vec F(const vec& x) return x%(x+1); // returns x.*(x+1)

vec x = {0,1,2};

vec gradient = grad(f,x); // should return [2*0, 2*1, 2*2] = [0,2,4]

mat Jac = approx_jacobian(F,x);
/* Jac = [1.0   0.0   0.0
          0.0   3.0   0.0
          0.0   0.0   4.0];
*/
```

Spectral Derivatives

Given function defined over an interval, we can approximate the derivative of the function with spectral accuracy using the `FFT`. The function `spectral_deriv()` does this:

```
poly_interp spectral_deriv(const function<double(double)>& f,
                        double a, double b,
                        unsigned int sample_points = 50);
```

We sample the function at chebyshev nodes scaled to the interval [a,b]; the function returns a polynomial object that can be evaluated anywhere on the interval. We can specify more sample points for more accuracy.

Example:

```
double f(double x) return sin(x*x);

auto df = spectral_deriv(f, -2, 2);

vec x = linspace(-2,2);
vec derivative = df(x);
```

For more discrete derivatives see `numerics::ode`.

Linear Root Finding and Optimization

Conjugate Gradient Method

Armadillo features a very robust `solve()` and `spsolve()` direct solvers for linear systems, but in the case where less precise solutions of very large systems (especially sparse systems) iterative solvers may be more efficient. The functions `cgd()` solve systems of linear equations $A\mathbf{x} = \mathbf{b}$ when A is symmetric positive definite (sparse or dense), or in the least squares sense $A^T A\mathbf{x} = A^T\mathbf{b}$ (dense only) by conjugate gradient method.

```
void cgd(arma::mat& A, arma::mat& b, arma::mat& x, double tol = 1e-3, unsigned int max_iter = 0);
```

if `max_iter <= 0`, then `max_iter = b.n_rows`.

note: when solving for `x` in `A*x = b`, if `A` is dense but not square or symmetric, the solver will set `b=A.t()*b` and `A = A.t()*A`, so `A` and `b` **will be modified outside the scope of the function**. The resulting system has worse conditioning.

note: in the sparse case, if A is not symmetric positive definite, the solver will quit and print an error message to `std::cerr`.

Adjusted Gradient Descent

Convex Constraint Linear Maximization

For solving linear ***maximization*** problems with linear constraints, we have the simplex algorithm that computes solutions using partial row reduction:

```
double simplex(arma::mat& simplex_mat, arma::vec& x);
double simplex(const arma::rowvec& F,
               const arma::mat& RHS,
               const arma::vec& LHS,
               arma::vec& x);
```

In the case that the user knows how to define the simplex matrix, we have the first definition. The function returns the ***maximum*** value within a convex polygon. The location of the maximum is stored in `x`.

Otherwise, the user can specify the linear function to ***maximize*** `f(x)`, say, by providing a row vector `F` such that `f(x)=F*x`. The constraints take the form: `RHS*x <=LHS`. The function returns the ***maximum*** value of `f(x)` that satisfies the constraints, and the location of the max is stored in `x`.

Nonlinear Root Finding

All of the nonlinear solver inherit from the `nlsolver` class:

```
class nlsolver {
    public:
    unsigned int max_iterations;
    double tol;
    int num_iterations();
    int get_exit_flag();
    int get_exit_flag(std::string& flag);
};
```

Where `max_iterations` lets you specify the maximum iterations allowed, the default is 100. The parameter `tol` is to specify a stopping criteria, the default is 1e-3. For most solvers the stopping criteria is $||x_{k+1} - x_k||_\infty <$ tol.

The member function `num_iterations()` returns the number of iterations actually needed by the solver.

The member function `get_exit_flag()` returns a 0, 1, or 2: * 0 : the solver successfully converged. * 1 : the maximum number of iterations was reached. * 2 : a NaN or Infinite value was encountered during a function evaluation.

If a string is provided, the function will overwrite the string with a description.

Newton's method

This is an implementation of Newton's method for systems of nonlinear equations. A jacobian function of the system of equations is required. As well as a good initial guess:

```
class newton : public nlsolver {
    public:
    bool use_cgd;
    newton(double tolerance = 1e-3);
    void fsolve(const function<arma::vec(const arma::vec&)>& f,
            const function<arma::mat(const arma::vec&)>& J,
            arma::vec& x,
            int max_iter = -1);
};
```

The class is initialized with the optional parameter to specify the tolerance for the stopping criteria.

The function `fsolve` solves the system $f(x) = 0$. The initial guess should be stored in `x` and this value will be updated with the solution found by Newton's method. The parameter `use_cgd` allows the user to specify where conjugate gradient method should be used to determine the update direction (or perform a direct solve), the default value is `true`. Specifying `max_iter` here will set the value `max_iterations` which has public scope. If `max_iter <= 0`, then `max_iterations` will not be changed.

There is also a single variable version:

```
double newton_1d(const function<double(double)>& f,
              const function<double(double)>& df,
              double x,
              double err = 1e-5);
```

Broyden's Method

This solver is similar to Newton's method, but does not require the jacobian matrix to be evaluated at every step; instead, the solver takes rank 1 updates of the inverse of the estimated Jacobian using the secant equations (wikipedia). Providing a jacobian function does improve the process, but this solver requires far fewer Jacobian evaluations than Newton's method. If none is provided the initial jacobian is computed using finite differencing (as opposed to being initialized to the identity) as this drastically improves the convergence.

```
class broyd : public nlsolver {
    public:
    broyd(double tolerance = 1e-3);
    void fsolve(std::function<arma::vec(const arma::vec&)> f,
                arma::vec& x,
                int max_iter = -1);
    void fsolve(std::function<arma::vec(const arma::vec&)> f,
                std::function<arma::mat(const arma::vec&)> jacobian,
                arma::vec& x,
                int max_iter = -1);
};
```

Levenberg-Marquardt Trust Region/Damped Least Squares

This solver performs Newton like iterations, replacing the Jacobian with a damped least squares version (wikipedia). The jacobian is updated with Broyden's rank 1 updates of the Jacobian itself (rather than the inverse), consequently it is recommended that a jacobian function be provided to the solver, otherwise the jacobian will be approximated via finite differences.

```
class lmlsqr : public nlsolver {
    public:
```

```
    double damping_param;
    double damping_scale;
    bool use_cgd;
    lmlsqr(double tolerance = 1e-3);
    void fsolve(std::function<arma::vec(const arma::vec& x)> f,
                arma::vec& x,
                int max_iter = -1);
    void fsolve(std::function<arma::vec(const arma::vec& x)> f,
                std::function<arma::mat(const arma::vec& x)> jacobian,
                arma::vec& x,
                int max_iter = -1);
};
```

The parameter `use_cgd` allows the user to specify where conjugate gradient method should be used to determine the update direction (or perform a direct solve), the default value is `true`.

The parameters `damping_param` and `damping_scale` directly affect how the algorithm determines the trust region. ### Fixed Point Iteration with Anderson Mixing Solves problems of the form $x = g(x)$. It is possible to solve a subclass of these problems using fixed point iteration i.e. $x_{n+1} = g(x_n)$, but more generally we can solve these systems using Anderson Mixing: $x_{n+1} = \sum_{i=p}^{n} c_i g(x_i)$, where $1 \leq p \leq n$ and $\sum c_i = 0$.

```
class mix_fpi : public nlsolver {
    public:
    unsigned int steps_to_remember;
    mix_fpi(double tolerance = 1e-3, uint num_steps = 5);
    void find_fixed_point(const std::function<arma::vec(const arma::vec&)>& g,
                            arma::vec& x,
                            int max_iter = -1);
};
```

We can specify to the solver how many previous iterations we want to remember i.e. we can specify $(n - p)$ using the `steps_to_remember` parameter.

fzero

Adaptively selects between secant method, and inverse interpolation to find a *simple* root of a single variable function in the interval `[a,b]`.

```
double fzero(const function<double(double)>& f, double a, double b, double tol = 1e-5);
```

Secant

Uses the secant as the approximation for the derivative used in Newton's method. Attempts to bracket solution for faster convergence, so providing an interval rather than two initial guesses is best.

```
double secant(const function<double(double)>& f, double a, double b, double tol = 1e-5);
```

Bisection method

Uses the bisection method to find the solution to a nonlinear equation within an interval.

```
double bisect(const function<double(double)>& f,
              double a, double b, double tol = 1e-2);
```

Nonlinear Optimization

fminbnd

provided a continuous function $f : (a, b) \to \mathbb{R}$ which is not necessarily continuous at the end points, we can find a local minimum of f within a small number of steps (bounded by $\approx 2.88[\log_2 \frac{b-a}{\epsilon}]^2 \approx 100$ function evaluations, when we select $\epsilon = 10^{-8} \times (b - a)$). The method:

```
double fminbnd(const function<double(double)>& f, double a, double b);
```

solves the problem: $\text{fminbnd}(f,a,b) = \text{argmin}_{x\in(a,b)} f$ using the algorithm provided by Brent (1972).

fminsearch

provided a continuous and finite function $f : \mathbb{R} \to \mathbb{R}$ which is not-necessarily continuous or finite at $\pm\infty$, we can attempt to find a local minimum of f near x_0 ussually within a small number of iterations (and likely to converge quickly for strongly convex f). The method:

```
double fminsearch(const function<double(double)>& f, double x0, double alpha=0);
```

solves the problem $\text{fminsearch}(f, x_0) = \text{argmin}_{\text{near } x_0} f$ using the Nelder-Mead algorithm restricted to one dimension. The parameter `alpha` specifies an initial step size for the algorithm in the positive direction. If one is not provided (or a non-positive value is provided) then $\alpha = \max\{\epsilon, \epsilon \times |x_0|\}$.

Multivariate Minimization

All of the nonlinear optimizers inherit from the class `optimizer`:

```
class optimizer {
    public:
    uint max_iterations;
    double tol;
    int num_iterations();
    int get_exit_flag();
    int get_exit_flag(std::string& flag);
};
```

Where `max_iterations` lets you specify the maximum iterations allowed, the default is 100. The parameter `tol` is to specify a stopping criteria, the default is 1e-3. For most solvers the stopping criteria is $||x_{k+1} - x_k||_\infty <$ tol.

The member function `num_iterations()` returns the number of iterations actually needed by the solver.

The member function `get_exit_flag()` returns a 0, 1, or 2: * 0 : the solver successfully converged. * 1 : the maximum number of iterations was reached. * 2 : a NaN or Infinite value was encountered during a function evaluation.

If a string is provided, the function will overwrite the string with a description.

Broyden–Fletcher–Goldfarb–Shanno algorithm

Uses the BFGS algorithm for minimization using the strong Wolfe conditions. This method uses symmetric rank 1 updates to the inverse of the hessian using the secant equation with the further constraint that the hessian remain symmetric positive definite.

```
class bfgs : public optimizer {
    public:
    double wolfe_c1;
    double wolfe_c2;
    double wolfe_scale;
    bool use_finite_difference_hessian;
    bfgs(double tolerance = 1e-3);
    void minimize(const std::function<double(const arma::vec&)>& f,
                  const std::function<arma::vec(const arma::vec&)>& grad_f,
                  arma::vec& x, int max_iter = -1);
    void minimize(const std::function<double(const arma::vec&)>& f,
                  const std::function<arma::vec(const arma::vec&)>& grad_f,
                  const std::function<arma::mat(const arma::vec&)>& hessian,
                  arma::vec& x, int max_iter = -1);
};
```

The function `minimize` solves the system $\min_x f(x)$. The initial guess should be stored in `x` and this value will be updated with the solution found by BFGS.

The parameter `use_finite_difference_hessian` allows the user to specify whether to approximate the initial hessian by finite diferences, the default value is `false`, so instead the initial hessian is set to the identity matrix.

The parameters `wolfe_c1`, `wolfe_c2` are the traditional paramters of the strong wolfe conditions: $f(x_k + \alpha_k p_k) \leq f(x_k) + c_1 \alpha_k p_k^T \nabla f(x_k)$ and $-p_k^T \nabla f(x_k + \alpha_k p_k) \leq -c_2 p_k^T \nabla f(x_k)$. The default values are `wolfe_c1 = 1e-4` and `wolfe_c2 = 0.9`.

The parameter `wolfe_scale` is used in the line minimization step to determing α_k satisfying the Wolfe conditions. We constrain `0 <= wolfe_scale <= 1`, a value closer to 1 allows for slower but potentially more accurate line minimization. The default value is 0.5.

note: like Broyden's method, `bfgs()` stores the inverse hessian in memory, this may become inneficient in space and time when the problem is sufficiently large.

Limited Memory BFGS

Uses the limited memory BFGS algorithm, which differs from BFGS by storing a limited number of previous values of `x` and `grad_f(x)` rather than a full matrix. The number of steps stored can be specified by `steps_to_remember`.

```
class lbfgs : public optimizer {
    public:
    unsigned int steps_to_remember;
    double wolfe_c1;
    double wolfe_c2;
    double wolfe_scale;
    lbfgs(double tolerance = 1e-3, uint num_steps = 5);
    void minimize(const std::function<double(const arma::vec&)>& f,
            const std::function<arma::vec(const arma::vec&)>& grad_f,
            arma::vec& x, int max_iter = -1);
};
```

Momentum Gradient Descent

Momentum gradient descent using adaptive line minimization.

```
class mgd : public optimizer {
    public:
    double damping_param;
    double step_size;
    mgd(double tolerance = 1e-3);
    void minimize(const std::function<arma::vec(const arma::vec&)>& grad_f,
                    arma::vec& x,
                    int max_iter = -1);
};
```

The parameter `damping_param` is has a good explaination found in this article. Setting this value to 0 is equivalent to traditional gradient descent.

The step size can be specified by `step_size` this can improve performance over the adaptive line minimization when the gradient is easy to evaluate but the may require more itterations until convergence.

Nonlinear Conjugate Gradient Descent

Uses the conjugate gradient algorithm for nonlinear objective functions with adaptive line minimization. Although this method uses only gradient information, it benefits from quasi-newton like super-linear convergence rates.

```
class nlcgd : public optimizer {
    public:
    double step_size;
    nlcgd(double tolerance = 1e-3);
    void minimize(const std::function<arma::vec(const arma::vec&)>& grad_f,
                    arma::vec& x,
                    int max_iter = -1);
};
```

The step size can be specified by `step_size` this can improve performance over the adaptive line minimization when the gradient is easy to evaluate but the may require more itterations until convergence.

Adjusted Gradient Descent

This method was designed to approximate the the Newton direction by storing only one previous step; the idea relies on the zig-zag behavior of traditional gradient descent. This method benefits from super-linear convergence rates similar to quasi-newton methods.

```
class adj_gd : public optimizer {
    public:
    double step_size;
    adj_gd(double tolerance = 1e-3);
    void minimize(const std::function<arma::vec(const arma::vec&)>& grad_f,
                    arma::vec& x,
                    int max_iter = -1);
};
```

The step size can be specified by `step_size` this can improve performance over the adaptive line minimization when the gradient is easy to evaluate but the may require more itterations until convergence.

note: I designed this method from experimentation, and it seems effective.

Nelder-Mead Gradient Free Minimization

The Nelder-Mead method is a derivative free method that constructs a simplex in n dimensions and iteratively updates its vertices in the direction where the function decreases in value. This method is ideal for low dimensional problems (e.g. 2,3, 10 dimensions, maybe not 100, though). The simplex is initialized using a guess x_0 of the argmin of the objective function, the second vertex in the simplex is $x_1 = x_0 + \alpha\vec{v}_1/||\vec{v}_1||$ where $\vec{v}_1$ is a random vector $\vec{v}_1 \sim N(0,1)$. Iteratively each other point is constructed $x_i = x_0 + \vec{\alpha}\vec{v}_i/||\vec{v}_i||$ where $\vec{v}_i^T\vec{v}_j = \delta_{ij}$ (i.e. a new orthogonal direction). This is done to guarantee the simplex is spread out, moreover the direction is random (as oposed to, e.g., the coordinate directions) to avoid pathological cases.

```
class nelder_mead : public optimizer {
    public:
    double step;
    double expand;
    double contract;
    double shrink;
    double initial_side_length;
    nelder_mead(double tolerance = 1e-2);

    void minimize(const std::function<double(const arma::vec&)>& f, arma::vec& x);
};
```

The parameter `step` is associated with the scaling of the reflection step, the default value is 1. The parameter `expand` is the scaling of the expanding step, the default value is 2. The parameter `contract` is the scaling of the contraction step, the default value is 0.5. The paramter `shrink` is the scaling of the shrinking step, the default value is 0.5. All of these parameters are explained here.

The parameter `initial_side_length` is the value α as desribed in the simplex initialization procedure discussed above.

Genetic Maximization Algorithm

This method uses a genetic algorithm for ***maximization***.

```
class gen_optim : public optimizer {
    public:
    double reproduction_rate;
    double mutation_rate;
    double search_radius;
    double diversity_weight;
    unsigned int population_size;
    unsigned int diversity_cutoff;
    unsigned int random_seed;
    unsigned int max_iterations;
    gen_optim(double tolerance = 1e-1);
    void maximize(const std::function<double(const arma::vec& x)>& f,
                    arma::vec& x,
                    const arma::vec& lower_bound,
                    const arma::vec& upper_bound);
    void maximize(const std::function<double(const arma::vec&)>& f, arma::vec& x);
};
```

This method has a variety of parameters for updating the population of parameters to minimize with respect to: * `population_size` : number of samples. * `reproduction_rate` : parameter for geometric probability distribution of "reproducing agents". i.e. if `reproduction_rate` is close to 1, then only the most fit will reproduce and the algorithm will converge more quickly at the cost of possibly not optimal results (such a getting stuck at local maxima). If `reproduction_rate` is close to 0, then most members will be able to participate at the cost of slower convergence. default value = 0.5. * `diversity_limit` : number of iterations after which we stop incentivising variance in the population. A lower value means quicker convergence at the cost possible not optimal results (such as getting stuck at local optima). * `mutation_rate` : rate at which to introduce random perturbation to the population. Values close to 1 result in a population with higher variance resulting in slower convergence. Values close to 0 result in a population with higher variance resulting in faster convergence at the cost possible not optimal results (such as getting stuck at local optima).

We can use this method for both box constrained and unconstrained maximization.

Interpolation

Cubic Interpolation

```
class cubic_interp
```

Fits piecewise cubic polynomials to data. The fitting occurs on construction or using the `fit` function:

```
cubic_interp::cubic_interp(const arma::vec& x, const arma::mat& Y);

cubic_interp& cubic_interp::fit(const arma::vec& x, const arma::mat& Y); // returns *this
```

We can save/load an interpolating object to a stream (such as a file stream):

```
cubic_interp::save(ostream& out);
cubic_interp::load(istream& in);
```

We can also load a saved object on construction:

```
cubic_interp::cubic_interp(istream& in);
```

Note, the data matrix will be stored to the stream as part of the object and can be recovered when the object is loaded using:

```
arma::vec cubic_interp::data_X(); // independent values
arma::mat cubic_interp::data_Y(); // dependent values
```

We can predict based on the interpolation using the `predict` member function or the `()` operator:

```
arma::mat cubic_interp::predict(const arma::vec&);
arma::mat cubic_interp::operator()(const arma::vec&);
```

Polynomial Interpolation

Class wrapper for armadillo's `polyfit` and `polyval` specialized for interpolation.

```
class poly_interp
```

We fit the polynomial on construction or using `fit()`:

```
poly_interp::poly_interp(const arma::vec& x, const arma::mat& Y);

poly_interp& poly_interp::fit(const arma::vec& x, const arma::mat& Y); // returns *this
```

We can save/load an interpolating object to a stream (such as a file stream):

```
poly_interp::save(ostream& out);
poly_interp::load(istream& in);
```

We can also load a saved object on construction:

```
poly_interp::poly_interp(istream& in);
```

Note, the data matrix will be stored to the stream as part of the object and can be recovered when the object is loaded using:

```
arma::vec poly_interp::data_X(); // independent values
arma::mat poly_interp::data_Y(); // dependent values
```

We can predict based on the interpolation using the `predict` member function or the `()` operator:

```
arma::mat poly_interp::predict(const arma::vec&);
arma::mat poly_interp::operator()(const arma::vec&);
```

If there is only a need to fit and interpolate a data set once, we may find it more efficient ($\mathcal{O}(n^3) \to \mathcal{O}(n^2)$) and numerically stable(!) to interpolate using Lagrange interpolation:

```
arma::mat lagrange_interp(const arma::vec& x,
                          const arma::mat& Y,
                          const arma::vec& xgrid,
                          bool normalize = false);
```

where `xgrid` is the set of values to interpolate over.

For high order polynomial interpolation, there is a likely hazard of exteme fluctuations in the values of polynomial (Runge's Phenomenon). We can atempt to address this problem in `lagrange_interp` by setting `normalize=true`. If the i^{th} Lagrange interpolating polynomial is $L_i(x) = \prod_{j\neq i} \frac{x-x_j}{x_i-x_j}$, then the interpolant is of the form: $f(x) = \sum_i y_i L_i(x)$. When `normalize=true`, the interpolant is instead: $\hat{f}(x) = \sum_i y_i L_i(x) e^{-(x-x_i)^2/\nu}$, where $\nu = \text{range}(x)/2n$. This normalization helps whenever x is approximately uniform.

note: When using `normalize=true`, remember that the resulting function is no longer a polynomial.

Sinc interpolation

Given sorted ***uniformly spaced*** points on an interval, we can interpolate the data using a linear combination of sinc functions $\text{sinc}(x) = \frac{\sin(\pi x)}{\pi x}$:

```
arma::mat sinc_interp(const arma::vec& x,
                      const arma::mat& Y,
                      const arma::vec& xgrid);
```

Data Science

K-Fold Train Test Split

When performing cross validation we may want to split the data into training and testing subsets. The `k_fold` procedure splits the data into `k` equal sized subsets of the data (randomly selected) for repeated training and testing.

```
class k_folds {
    public:
    k_folds(const arma::mat& x, const arma::mat& y, unsigned int k=2, unsigned int dim=0);
    arma::mat test_set_X(unsigned int j);
    arma::mat test_set_Y(unsigned int j);
    arma::mat train_set_X(unsigned int j);
    arma::mat train_set_Y(unsigned int j);
};

class k_fold_1d {
    public:
    k_folds(const arma::mat& x, unsigned int k=2, unsigned int dim=0);
    arma::mat test_set(unsigned int j);
    arma::mat train_set(unsigned int j);
};
```

The parameter `dim` informs over which dimension of `x,y` to split over, if `dim` is 0, then we split up the columns, if `dim` is 1, then we split up the rows.

we access the `j`th fold using `fold_X` and `fold_Y` and we access the complement to the `j`th fold using `not_fold_X` and `not_fold_Y`. We can alternatively use the `[]` operator has the functionality both `fold_X` using positive indexing and `not_fold_X` using negative indexing. We can use the `()` operator as both `fold_Y` using positive indexing and `not_fold_Y` using negative indexing.

example:

```
mat X = randu(100,1);
mat Y = 2*x;

k_fold train_test(X,Y,3);

int j = 0;
mat train_X =  train_test.train_set_X(j);
mat test_X = train_test.test_set_X(j);

mat train_Y = train_test.train_set_Y(j);
mat test_Y = train_test.test_set_Y(j);
```

Data Binning

Whenever data is sampled from continuous features, it is sometimes useful (and easy) to bin the data into uniformly spaced bins. A primary benefit is reducing the complexity to $O(1)$ for any computation (number of bins is fixed by the user). In the univariate case, 500 bins is more than sufficient for large data.

The following class produces bins for univariate data.

```
class bin_data
```

we initialize the object:

```
bin_data::bin_data(unsigned int num_bins=0);
```

When `num_bins` is not specified, it is selected during call to `to_bins()`. It will be set to 500 if the data size $n > 1000$, or $n/10$ if $n/10 > 30$, or $n/5$ in all other cases.

The `bin_data` object can bin two varieties of data:

```
void to_bins(const arma::vec& x);
```

In this case x is stratified into uniformly spaced bins in the range $[\min(x) - \epsilon, \max(x) + \epsilon]$ (for a small $\epsilon > 0$) which has an associated spacing δ. To each bin we associate a "count" which is computed by finding every point in x that is within $\pm\delta$ of that bin and weighing the count by 1 minus the normalized linear distance, i.e.:

for each $x_i \in \text{bin}(j) \pm \delta$: $\text{bin}(j).count \mathrel{+}= 1 - \frac{|\text{bin}(j) - x_i|}{\delta}$

note: most likely $\text{bin}(j).count$ is not an integer but it is guaranteed that $\sum_{j=1}^{n} \text{bin}(j).count = n$.

We can also bin data with an associated univariate response variable:

```
void to_bins(const arma::vec& x, const arma::vec& y);
```

In this case, x is similarly stratified, but the counts are computed by scaling the counts by the response variable y. i.e.:

1. for each $x_i \in \text{bin}(j) \pm \delta$:
    1. $w_i = 1 - \frac{|\text{bin}(j) - x_i|}{\delta}$
    2. $\text{bin}(j).count \mathrel{+}= w_i * y_i$
2. $\text{bin}(j).count \mathrel{/}= \sum_i w_i$

Either way, we access our bins and counts by referencing the member variables:

```
const vec& bins;
const vec& counts;
```

we can also reference the variables:

```
const int& n_bins;
const double& bin_width;
```

example:

```
vec x = randn(200);
vec y = exp(-x);

int n_bins = 20;
bin_data distribution(n_bins);
distribution.to_bins(x);
vec x_bins = distribution.bins;
vec x_counts = distribution.counts;
// if we plot x_bins and x_counts we should expect a bell shape

bin_data discretized_response(n_bins);
discretized_response.to_bins(x,y);
x_bins = discretized_response.bins;
y_discrete = discretized_response.counts;
// if we plot x_bins and y_discrete we should expect a decaying exponentional
```

K-Means Clustering

```
class kmeans
```

we can cluster unlabled data using the kmeans algorithm. On construction, we must specify our data and the number of clusters; our data is provided as an `arma::mat` where each row is a data entry.

```
kmeans::kmeans(arma::mat& data, int num_clusters);
```

We can save and load a `kmeans` object to file using stream objects (note, the data matrix will be stored to the stream as part of the object and can be recovered when the object is loaded).

```
kmeans::save(ostream& out);

kmeans::load(istream& in);
kmeans::kmeans(istream& in); // can be loaded in on construction
```

Once our `kmeans` object is constructed we can get information about which data went to which cluster with `get_clusters()` and we can get the cluster means themselves using `get_centroids()`:

```
arma::vec kmeans::get_clusters() const;
/*
the i^th elem corresponds to the i^th data point.
The i^th element contains the cluster number which is an int ranging on [0,k-1]
*/

arma::mat kmeans::get_centroids() const;
/*
the i^th row is the i^th cluster where i is ranging on [0, k-1]
/*
```

We can test our clustering by predicting the cluster of data the `kmeans` was not trained on, we can use either the `()` operator or `predict()`:

```
arma::rowvec kmeans::operator()(const arma::mat& X);
arma::rowvec kmeans::predict(const arma::mat& X);
/*
the i^th elem corresponds to the i^th data point.
The i^th element contains the cluster number which is an int ranging on [0,k-1]
*/

int kmeans::operator()(const arma::rowvec& X);
int kmeans::predict(const arma::rowvec& X);
/*
output in [0,k-1]
*/
```

We can also ask `kmeans` to give us all the data from our original data belonging to a specific cluster. We can do this with the `[]` operator or `all_from_cluster()`:

```
arma::mat operator[](int i);
arma::mat all_from_cluster(int i);
```

We can print a summary of our `kmeans` clustering to a stream:

```
ostream& kmeans::summary(ostream& out = cout);
```

We can also retrieve basic documentation for the class using the `kmeans::help(ostream& out = cout)` member function.

Splines

Fit radial basis splines to any dimensional data set. The construction is based on both multivariate polynomial terms and a radial basis kernel (polyharmonic). We regularize the fit by constraining the magnitude of the polyharmonic basis. Essentially we are solving the quadratic optimization problem: $\min_{c,d} ||y - Kc - Pd||_2^2 + \lambda||c||_2^2 + \gamma||d||_1$. The parameter $\gamma \geq 0$ is always determined by cross-validation, the intention is to reduce the total number of polynomial terms. The parameter $\lambda \geq 0$ can be provided before fitting, or can be determined automatically by cross validation (when $\lambda = 0$, the resulting function interpolates. When $\lambda \to \infty$ the resulting function tends toward the polynomial fit). The cross validation is done efficiently by computing the eigenvalue decomposition of K once and solving each regularized problem using matrix multiplication which improves the overall complexity from $\mathcal{O}(kN^3)$ (i.e. k Cholesky decompositions) to $\mathcal{O}(N^3 + kN^2)$ (i.e. k matrix-vector multiplications).

```
class splines
```

We construct the object via any of the following:

```
splines::splines(int poly_degree=1);
```

Where `poly_degree` is the degree of the polynomial, and it should be noted, that the size of the system grows exponentially with `poly_degree` and becomes extremely ill conditioned for large values. It is recomended that `1 <= poly_degree <= 3`.

The functions:

```
splines::set_smoothing_param(double lambda);
splines::set_degrees_of_freedom(double df);
```

are used to set the smoothing parameter before fitting. The function `set_smoothing_parameter` sets this value directly, while `set_degrees_of_freedom` controls the value by root-finding using the relationship: $df(\lambda) = \sum_i \frac{d_i^2}{d_i^2+\lambda}$ where d_i are the eigenvalues of K the smoothing kernel matrix.

If neither functions are called, the smoothing parameter will be determined from cross validation.

We fit the splines object using:

```
void fit(const arma::mat& X, const arma::mat& Y);
```

We can predict based on our fit using the following functions:

```
arma::mat splines::predict(const arma::mat& X);
arma::mat splines::operator()(const arma::mat& X);
```

We can extract a variety of additional information from our fit:

```
const double& smoothing_param; // lambda
const double& eff_df; // effective degrees of freedom
const double& RMSE; // root mean squared error
const arma::mat& residuals;
const arma::mat& poly_coef; // coefficients for polynomials
const arma::mat& rbf_coef; // coefficients for splines
const arma::mat& data_X; // independent variables
const arma::mat& data_Y; // dependent variables
const arma::vec& rbf_eigenvals; // eigenvalues of K
const arma::mat& rbf_eigenvecs; // eigenvectors of K

arma::mat splines::eval_rbf(const arma::mat&);
arma::mat splines::eval_poly(const arma::mat&);
```

We can save and load a splines object to a stream (file):

```
void splines::save(ostream& out);

void splines::load(istream& in);
splines::splines(istream& in); // load on construction
```

Logistic Regression

Fit linear or kernel logistic model to any dimensional data set (logistic or ridge classification). By default we fit gaussian kernel basis with radius 1 along side linear basis, but just an option exists to remove the radial basis. By default the regression is done via L2-regularization and the regularization parameter is determined by cross validation, the metric for the cross validation and for the optimization procedure is log-likelihood. The resulting model is a one vs. all probabilistic model, when the categories are predicted the maximum probability is selected. When fitting the full rbf model, `lbfgs` solver is used, but when fitting just the linear basis, `nelder_mead` is used instead.

```
class logistic_regression
```

We initialize the class by specifying β which is the radius of the gaussian, i.e. $K(x,x_0) = e^{-||x-x_0||^2/\beta}$, and the regularizing paramter `Lambda`:

```
logistic_regression::logistic_regression(double Beta = 1, double Lambda = nan)
```

if `Beta <= 0`, then only a linear basis is used, if `Lambda < 0 || isnan(Lambda)` then it is determined by cross validation.

We fit the model by calling `fit()`:

```
void logistic_regression::fit(const arma::mat& x, const arma::mat& y);
```

The parameter `x` is simply the independent variable, where each row is an observation. The parameter `y` on the other hand should have as many colomns as categories and the (i,j) element should be 1.0 if the i^th observation belongs to the j^th category, and 0.0 otherwise.

Once our logistic model is fit, we can predict probabilities and categories using:

```
arma::mat predict_probabilities(const arma::mat& xx);
arma::umat predict_categories(const arma::mat& xx);
```

The output of `predict_categories()` is of the same format as the `y` matrix from fitting. The (i,j) element of `predict_probabilities()` specifies the probability that the i^th observation belongs to the j^th category.

We can extract extra information from the fit such as the various parameters and coefficients computed and we can get the radial basis matrix for any new sample data:

```
double logistic_regression::kernel_param() const; // beta
double logistic_regression::regularizing_param() const; // lambda

/* z = [1, X]*c + K(X)*d  ==> probability = softmax(z) */
arma::mat logistic_regression::rbf(const arma::mat& xx); // K(X) radial basis kernel
arma::mat logistic_regression::linear_coefs() const; // c
arma::mat logistic_regression::kernel_coefs() const; // d
```

The `logistic_regression` object also saves a copy of the data which can be retrieved:

```
arma::mat logistic_regression::data_X();
arma::mat logistic_regression::data_Y();
```

We can also get a table of the results from cross validation:

```
arma::mat logistic_regression::get_cv_results() const;
```

Where the first colomn is the set of `lambda`s tested, and the second colomn is the coresponding average log-likelihood score from cross validation, the value lambda used by the solver will be the one with the maximum log-likelihood associated with it.

We can save/load `logistic_regression` objects to stream (files) via:

```
void logistic_regression::load(std::istream& in);
void logistic_regression::save(std::ostream& out);
```

or on construction:

```
logistic_regression::logistic_regression(std::istream& in);
```

k Nearest Neighbors Regression

Predict on data using k nearest neighbors. The method applies both kd-trees and a brute force strategy to compute the nearest neighbors. The kd-tree implementation is the default if the dimension of the space is relatively small and there is sufficient data to justify it. If this is not the case, the algorithm will instead default to a brute force strategy.

When predicting there are two options (1) average value of the k nearest neighbors or (2) a distance weighing of the nearest neighbors using a radial basis function $w_i = e^{-||x-x_i||^2/\sigma^2}/\sum_j e^{-||x-x_j||^2/\sigma^2}$. The default method is averaging. The number of nearest neighbors can be defined explicitly or a set of integers can be provided and the data structure will determine the optimal value by cross validation.

```
class knn_regression
```

To construct a knn object, we must supply either `k` the number of NNs or a vector of `k` values to test via cross validation on the MSE score.

```
enum class knn_algorithm {
    AUTO,
    KD_TREE,
    BRUTE
};

enum class knn_metric {
    CONSTANT,
    DISTANCE
};

knn_regression::knn_regression(unsigned int k, knn_algorithm alg, knn_metric metr);

knn_regression::knn_regression(const arma::uvec& k, knn_algorithm alg, knn_metric metr);
```

We fit the object via:

```
knn_regression& knn_regression::fit(const arma::mat& x, const arma::mat& y);
```

The parameter `x` is the independent variable, where each row is an observation. The parameter `y` is the dependent variable.

Once our model is fit, we can predict values of our function using:

```
arma::mat knn_regression::predict(const arma::mat& xx);
```

We can also request additional information from the object:

```
const int& knn_regression::num_neighbors; // get k used to predict
```

We can use a modified version of `knn_regression` for (multi-)classification:

```
class knn_classifier : public knn_regression
```

This class is treated the same as `knn_regression` with the additional functionality:

```
arma::mat knn_classifier::predict_probabilities(const arma::mat& xx);
arma::umat knn_classifier::predict_categories(const arma::mat& xx);
```

Where `predict_probabilities()` is equivalent to `predict()`. Additionally cross validation is performed on the F1 score instead of MSE.

Kernel Smoothing

Kernel smoothing may be applied to quickly approximate a function at a point x_0 by weighted sum of samples within a bandwidth β of x_0. The weights are determined by a symmetric positive definite kernel function $K(\cdot,\cdot)$ of which there are variety of options (we define $K_\beta(x,x_0) = k\left(r = \frac{||x-x_0||}{\beta}\right)$):

- `RBF`: $k(r) = \frac{1}{\sqrt{2\pi}}e^{-r^2/2}$
- `square`: $k(r) = 0.5 I_{r\leq 1}(r)$
- `triangle`: $k(r) = (1-r)I_{r\leq 1}(r)$
- `parabolic`: $k(r) = \frac{3}{4}(1-r^2)I_{r\leq 1}(r)$

Choosing β may be difficult, so, by default, the bandwidth will be determined by cross-validation.

The final estimate of the function looks like: $\hat{y}(x) = \frac{\sum_{i=1}^{n} y_i K_\beta(x,x_i)}{\sum_{i=1}^{n} K_\beta(x,x_i)}$

We can make kernel smoothing more efficient (effectively $O(1)$) for large data by first binning the data (linear binning). We compute the kernels with respect to the bins rather than the observations without reducing the quality of fit significantly. For large data sets Gramacki (2018) argues that 400-500 bins is almost always sufficient for univariate distributions. By default, when binning is requested and $n > 1000$ the number of bins defaults to 500 (otherwise, it is selected to be $n/10$ if $n/10 > 30$ or $n/5$ for all other cases).

```
class kernel_smooth
```

We construct a smoothing object via:

```
enum class kernels {
    gaussian,
    square,
    triangle,
    parabolic
};

kernel_smooth::kernel_smooth(kernels k=gaussian, bool binning=false);
kernel_smooth::kernel_smooth(double bdw, kernels k=gaussian, bool binning=false);
```

where `bdw` $= \beta$. Whenever `bdw` is not specified, we use cross validation.

We fit the object:

```
kernel_smooth& kernel_smooth::fit(const arma::vec& x,
                                  const arma::vec& y);

arma::vec kernel_smooth::fit_predict(const arma::vec& x,
                                     const arma::vec& y);
```

where `obj.fit_predict(x,y)` is equivalent to `obj.fit(x,y).predict(x)`.

We can predict based on our fit accroding:

```
arma::mat kernel_smooth::predict(const arma::mat& xgrid);
arma::mat kernel_smooth::operator()(const arma::mat& xgrid);

arma::mat kernel_smooth::predict(double xval);
arma::mat kernel_smooth::operator()(double xval);
```

The functions, `predict` and the operator `()` are equivalent.

The object retains additional information from fitting,

```
const arma::vec& data_x;
const arma::vec& data_y;
const double& bandwidth;
```

The kernel smoothing object may be saved to a file, and loaded from one:

```
void kernel_smooth::save(const std::string& fname);

void kernel_smooth::load(const std::string& fname);
kernel_smooth::kernel_smooth(const std::string& fname); // load on construction
```

Kernel Density Estimation

Smoothing kernels may be applied to density estimation as well. We define $K(\cdot,\cdot)$ same as for `kernel_smooth` only now we estimate the density for a single variable by: $\hat{f}(x) = \frac{1}{n\beta}\sum_{i=1}^{n} K_\beta(x,x_i)$.

Note we defined the kernels above so that $\int_{-\infty}^{\infty} \hat{f}(x)dx = 1$ and $\hat{f}(x) \geq 0$ i.e. a valid density function.

In this case, too, selecting a good bandwidth β is not trivial, and since the true density at the sampled values is not known, performing cross-validation is not as straight forward. Consequently we have four well known methods for bandwidth selection:

1. Rule of Thumb (1) : $\beta = 1.06sn^{-1/5}$, this method is optimal whenever the data is sampled from a normal distribution.
2. Rule of Thumb (2) : $\beta = 0.9\min(s, \frac{IQR}{1.34})n^{-1/5}$, this method is similar to (1) but is better at handling non-symmetric and bi-model distributions. This method is the most commonly used default estimate, and ussually produces good results.
3. Direct Plug-in : β is computed to minimize the asymptotic expansion of the mean integrated squared error $(\text{MISE} = \int_{-\infty}^{\infty}(\hat{f}_\beta(x) - f(x))^2 dx)$. The plug-in method is iterative, but for most applications 2 iterations are sufficient, and that is the version implemented here.
4. Grid search cross validation : β is sampled in the range $[0.05s, \text{range}(x)/4]$ using log-spacing. We compute an optimal β by approximating the RMSE, where the true density is estimated by a pilot density. The pilot density is computed using the entire data set and the second rule of thumb for the bandwidth. Then an RMSE for each bandwidth is computed by comparing the pilot density to the new density in line using training and testing sets.

The default method is (2) as it only requires computing descriptive statistics.

```
class kde
```

We construct a kernel density object via:

```
enum class bandwidth_estimator {
    rule_of_thumb_sd,
    min_sd_iqr,
    direct_plug_in,
    grid_cv
};

kde::kde(kernels k=gaussian, bandwidth_estimator method=min_sd_iqr, bool binning=false);
kde::kde(double bdw, kernels k=gaussian, bool binning=false);
```

where `bdw` $= \beta$.

we fit the density:

```
kde& kde::fit(const arma::vec& x);
```

we can predict densities for new query points:

```
arma::vec kde::predict(const arma::vec& xgrid);
arma::vec kde::operator()(const arma::vec& xgrid);

double kde::predict(double x);
double kde::operator()(double x);
```

We can sample our density (bootstrapping):

```
arma::vec kde::sample(unsigned int n=1);
```

The object retains additional information from fitting,

```
const arma::vec& data;
const double& bandwidth;
```

The kde object may be saved to a file, and loaded from one:

```
void kde::save(const std::string& fname);

void kde::load(const std::string& fname);
kde::kde(const std::string& fname); // load on construction
```

Ridge Regression

When fitting a large basis set to data (often the case in non-parametric modeling), overfitting becomes a significant problem. To combat this problem we can regularize our parameters during the fit. Essentially we are solving the minimization problem: $\min_c ||y - Xc||^2 + \lambda||c||_2^2$. Where $\lambda \geq 0$ is the regularization parameter and is determined from cross validation. When $\lambda = 0$, the fit tends toward high variance. When $\lambda \to \infty$, the fit tends toward high bias. Determining λ may be achieved by cross validation, like the spline class we take advantage of the (symmetric) eigenvalue decomposition (of $X^T X$) to speed up computations of the regularized solution. Cross-validation is performed using the generalized cross-validation score which approximates LOOCV.

```
class ridge_cv
```

With constructor:

```
ridge_cv::ridge_cv();
```

The object is fitted using:

```
void ridge_cv::fit(const arma::mat& X, const arma::mat& Y);
```

We get the result from fitting and some extra information:

```
const arma::mat& coef; // solution
const arma::mat& residuals;
const arma::mat& cov_eigvecs; // eigenvectors of X'*X
const arma::vec& cov_eigvals; // eigenvalues of X'*x
const double& regularizing_param; // optimal lambda
const double& RMSE; // root mean squared error
const double& eff_df; // effective degrees of freedom
```

LASSO Regression

Like Ridge regression, LASSO attempts to account for overfitting by computing penalizing large values of the coefficients. The objective is to solve the minimization problem $\min_c ||y - Xc||^2 + \lambda||c||_1$. Where $\lambda \geq 0$ is the regularization parameter and is determined by cross-validation. When $\lambda = 0$, the fit tends toward high variance. When $\lambda \to \infty$, the fit tends toward high bias. LASSO has the additional benefit of encoding sparsity, i.e. as λ increases, parameters tend to attain zero values; as such LASSO is often used for model reduction, where coefficients below a certain threshold are dropped from the model. The cross-validation performed to determine λ uses 2-Fold cross-validation, the coefficient vector c from one value of λ is used to initialize c for the next value of λ which leads to faster convergence of the optimization procedure.

```
class lasso_cv
```

With constructor:

```
lasso_cv::lasso_cv(double tol=1e-5, uint max_iter=1000);
```

where `tol` is the relative tolerance for the optimization procedure and `max_iter` is the maximum number of iterations for the procedure, this is applied to every call of the optimization function (`coordinate_lasso`). The number times the optimization is performed is bounded by $\mathcal{O}([\log_2(\frac{1}{tol})]^2)$ relying on the continuity of the $MSE(\lambda)$ of the fit.

The object is fitted using:

```
void lasso_cv::fit(const arma::mat& X, const arma::mat& Y, bool first_term_intercept=false);
```

Where `first_term_intercept` is used to specify whether the first column of `X` is an intercept term (i.e. [1,1,...,1]). If `first_term_intercept==true` then shrinkage will not be applied to $c[0]$.

We get the result from fitting and some extra information:

```
const arma::mat& coef; // solution
const arma::mat& residuals;
const double& regularizing_param; // optimal lambda
const double& RMSE; // root mean squared error
const double& eff_df; // effective degrees of freedom
```

This class is essentially just a wrapper for the function:

```
int coordinate_lasso(const arma::mat& y,
                     const arma::mat& X,
                     arma::mat& c,
                     double lambda,
                     bool first_term_intercept,
                     double tol,
                     uint max_iter,
                     bool verbose=false);
```

which solves $min_c||y - Xc||_2^2 + \lambda||c||_1$ for a specific λ. All the parameters are as above. The parameter `verbose`, when set to `true`, prints the optimization results per iteration. The function modifies `c` inplace, and returns `0` if it converged successfully, or `1` if the number of iterations reached `max_iter` before $||\Delta c||_\infty$ could fall below tol × $||c||_\infty$.

`numerics::ode` Documentation

Table of Contents

The following are all member of namespace `numerics::ode`.

Differentiation Operators

Given an interval $\Omega = [L, R]$, if we sample Ω at points $x = \{x_1, \cdots, x_N\}$ we can approximate the continuous operator $\frac{d}{dx}$ at the sampled points with discrete operator D. This operator can be applied to any differentiable $f : \Omega \to \mathbb{R}$ given the function values at the sample points: $y = \{f(x_1), \cdots, f(x_N)\}$ according to: $f'(x) \approx Dy$.

```
void diffmat4(arma::mat& D,
              arma::vec& x,
              double L, double R,
              unsigned int sample_points);
void diffmat2(arma::mat& D,
              arma::vec& x,
              double L, double R,
              unsigned int sample_points);
void cheb(arma::mat& D,
          arma::vec& x,
          double L, double R,
          unsigned int sample_points);
```

```
void cheb(arma::mat& D, arma::vec& x, unsigned int sample_points);
```

In all of these functions the discrete operator is assigned to `D`, and the sample points are assigned to `x`. The parameters `L` and `R` define the end points of the interval Ω. The parameter `sample_points` defines how many points to sample from the interval.

The function `diffmat4` samples the interval uniformly and provides a fourth order error term i.e. error is $\mathcal{O}(N^{-4})$. The resulting operator has a bandwidth of 4. It is also the case that the eigenvalues of D are all of the form $\lambda_k = -b_k i$ where $b_k \geq 0$ and $i = \sqrt{-1}$.

The function `diffmat2` samples the interval uniformly and provies a second order error term i.e. error is $\mathcal{O}(N^{-2})$. The resulting operator has a bandwidth of 2. It is also the case that the eigenvalues of D are all of the form $\lambda_k = -b_k i$ where $b_k \geq 0$ and $i = \sqrt{-1}$.

The function `cheb` samples the interval at Chebyshev nodes and converges spectrally. If no interval is provided, the interval is set to $[-1, 1]$. The resulting operator is dense. Moreover $(D_{\text{cheb}})^k = \frac{d^k}{dx^k}$.

In all three cases the $n \times n$ operator has rank $n - 1$ which follows from the intuition that the null space of the derivative is the set of all (piecewise-)constant functions.

A more generic differentiation matrix is offered by the following two functions:

```
arma::rowvec diffvec(const arma::vec& x, double x0, unsigned int k=1);

diffmat(arma::mat& D, const arma::vec& x, unsigned int k=0, unsigned int bdw=2);

diffmat(arma::sp_mat& D, const arma::vec& x, unsigned int k=0, unsigned int bdw=2);
```

Where `diffvec` returns a rowvector $\vec{d_k}$ such that $\vec{d_k} \cdot f(\vec{x}) \approx f^{(k)}(x_0)$.

The function `diffmat` produces a differentiation matrix D for any grid of points x (does not need to be uniform or sorted) such that $D_k f(x) \approx f^{(k)}(x)$. Setting the `bdw` parameter will allow the user to select the number of nearby points to use in the approximation, for example if `bdw=2` then for x_i, the points $\{x_{i-1}, x_i, x_{i+1}\}$ will be used in the approximation. Moreover, expect the error in the approximation to be $\mathcal{O}(h^{\text{bdw}})$ where h is the maximum spacing in x. A special benefit of `diffmat` is that we can find any order derivative, moreover for any $n \times n$ D_k we have $\text{rank}(D_k) = n - k$, and the eigenvalues are of the form $\lambda = i^k b$ where $b \geq 0$ and $i = \sqrt{-1}$. (so D_2 is positive semi-definite for example).

Given a linear ODE of the form: $y' + \alpha y = f(x)$ and the initial condition: $y(L) = \beta$, we can approximate the solution by solving the linear system: $(D + \alpha I)y = f(x) \wedge y(L) = \beta$. This can be solved by forward substituting $y(L) = \beta$ into the linear system and solving the rest of the system:

```
vec f(vec& x) {
    // do something
}
mat D;
vec x, y;
double L, R, alpha, beta;
int N;
int method; // set to 1 or 2

diffmat2(D,x,L,R,N); // or diffmat4(), or cheb(), or diffmat()

mat A = D.rows(1,N-1).cols(1,N-1) + alpha*eye(N-1,N-1);
vec d0 = D.col(0);
vec F = f(x.rows(1,N-1)) - d0.rows(1,N-1)*beta;

vec y(N);
y.rows(1,N-1) = solve(A,F);
```

If we have a system of m ODEs, we can solve both initial value problems and boundary value problems using a similar method where instead the operator is replaced with $(D \otimes I_{m,m})$ ($\otimes$ is the Kronecker product) and $f(x)$ is vectorized (if `F` is $n \times m$ then set `F = vectorise(F.t())`. Once a solution y is found it is reshaped so that it is $n \times m$ (if `y` is $nm \times 1$, then set `y = reshape(y,m,n).t()`). For these higher order system both initial value problems and boundary value problems may be solved.

Initial Value Problem Solvers

We define a system of initial value problem as having the form: $u' = f(t,u)$ with $u(0) = u_0$. Where t is the independent variable and $u(t)$ is the dependent variable and is a row vector. All of the systems solvers are able to handle events. Some of the solvers have error control via adaptive step size selection. For the implicit solvers we can also provide a jacobian matrix $\frac{\partial f}{\partial u}$ to improve solver performance. All implicit solvers use Broyden's method or Newton's method to compute steps.

All solver inherit from the `ivp` class:

```
class ivp {
    public:
    unsigned int max_nonlin_iter;
    double max_nonlin_err;
    unsigned int stopping_event;

    void add_stopping_event(const std::function<double(double,const arma::rowvec&)>& event,
                            event_direction dir = ALL);
};
```

Events are defined in a later section.

All IVP solvers have a function `ode_solve` which, at a higher level, describes the problem to solve and the solution (recall we are solving $u' = f(t,u), u(t_0) = u_0$):

```
void ode_solve(f [,J], t, U);
```

The parameter `f` is $f(t,u)$.

If `J` is provided, it is the jacobian matrix: $J(t,u) = \frac{\partial f}{\partial u}$. The jacobian may be provided to the implicit solvers, though they perform comparably without.

The parameter `t` must be initialized to `{t_initial, t_final}`, this parameter will be overwritten with the grid points selected by the solver on the interval.

The parameter `U` should be initialized to u_0 as a single row vector. This parameter will be overwritten by the solver so that the pair {`t(i), U.row(i)`} is $\{t_i, u(t_i)\}$.

Dormand-Prince 4/5

Fourth order explicit Runge-Kutta solver with adaptive step size for error control.

```
class rk45 : public ivp {
    public:
    double adaptive_step_min; // 0.05
    double adaptive_step_max; // 0.5
    double adaptive_max_err; // tol
    rk45(double tol = 1e-3);
    void ode_solve(
        const std::function<arma::rowvec(double,const arma::rowvec&)>& f,
        arma::vec& t,
        arma::mat& U);
};
```

Runge-Kutta Fourth Order

classical Fourth order explicit Runge-Kutta solver with constant step size.

```
class rk4 : public ivp {
    public:
    double step; // step_size
    rk4(double step_size = 0.1);
    void ode_solve(
        const std::function<arma::rowvec(double,const arma::rowvec&)>& f,
        arma::vec& t,
        arma::mat& U);
};
```

Runge-Kutta Implicit Fourth Order

Fourth order diagonally implicit Runge-Kutta solver with adaptive step size controlled via a third order approximation. Method is A-stable and L-stable.

```
class rk45i : public ivp {
    public:
    double adaptive_step_min;
    double adaptive_step_max;
    double adaptive_max_err; // tol
    rk45i(double tol = 1e-3);
    void ode_solve(
        const std::function<arma::rowvec(double,const arma::rowvec&)>& f,
        arma::vec& t,
        arma::mat& U);
    void ode_solve(
        const std::function<arma::rowvec(double,const arma::rowvec&)>& f,
        const std::function<arma::mat(double,const arma::rowvec&)>& jacobian,
        arma::vec& t,
        arma::mat& U);
};
```

This object accepts a jacobian matrix.

Runge-Kutta Implicit Fifth Order

Fifth order semi-implicit Runge-Kutta solver with constant step size. Method is A-stable and L-stable.

```
class rk5i : public ivp {
    public:
    double step;
    rk5i(double step_size = 0.1);
    void ode_solve(
        const std::function<arma::rowvec(double,const arma::rowvec&)>& f,
        arma::vec& t,
        arma::mat& U);
    void ode_solve(
        const std::function<arma::rowvec(double,const arma::rowvec&)>& f,
        const std::function<arma::mat(double,const arma::rowvec&)>& jacobian,
        arma::vec& t,
        arma::mat& U);
};
```

This object accepts a jacobian matrix.

Backwards Euler

First order implicit Euler's method with constant step size. (Euler's method is not accurate but very stable):

```
class am1 : public ivp {
    public:
    double step; // step_size
    am1(double step_size = 0.1);
    void ode_solve(
        const std::function<arma::rowvec(double,const arma::rowvec&)>& f,
        arma::vec& t,
        arma::mat& U);
    void ode_solve(
        const std::function<arma::rowvec(double,const arma::rowvec&)>& f,
        const std::function<arma::mat(double, const arma::vec&)>& jacobian,
        arma::vec& t,
        arma::mat& U);
};
```

This object accepts a jacobian matrix.

Adams-Moulton Second Order

Second order implicit linear multistep method with constant step size. (Will likely be replaced by a B-stable alternative)

```
class am2 : public ivp {
    public:
    double step; // step_size
    am2(double step_size = 0.1);
    void ode_solve(
        const std::function<arma::rowvec(double,const arma::rowvec&)>& f,
        arma::vec& t,
        arma::mat& U);
    void ode_solve(
        const std::function<arma::rowvec(double,const arma::rowvec&)>& f,
        const std::function<arma::mat(double, const arma::rowvec&)>& jacobian,
        arma::vec& t,
        arma::mat& U);
};
```

This object accepts a jacobian matrix.

IVP Events

The function:

```
enum class event_direction {
    NEGATIVE = -1,
    ALL = 0,
    POSITIVE = 1
};
void add_stopping_event(const std::function<double(double,const arma::rowvec&)>& event,
                        event_direction dir);
```

Allows the the user to add an event function which acts as a secondary stopping criterion for the solver. An event function specifies to the solver that whenever event$(t_k, u_k) = 0$ the solver should stop. We can further constrain the stopping event by controlling the sign of event(t_{k-1}, u_{k-1}). e.g. if `dir = NEGATIVE`, the solver will stop iff: event$(t_k, u_k) = 0$ ***and*** event$(t_{k-1}, u_{k-1}) < 0$.

The `ivp` member `stopping_event` will be set to the event index (of the events added) that stopped it. e.g. if the third event function added stops the solver, then `stopping_event = 2`.

Boundary Value Problems Solver

We can solve boundary value problems using finite difference methods. A procedure for simple linear problems was described in the operators section, but the following methods are far more generalized. Our problem is defined as follows:

Given interval domain $\Omega = [L, R]$, and ***system*** of ODEs $u' = f(x, u)$ with boundary conditions $g(u) = 0$ on $\partial\Omega$ which is equivalently defined: $g(u(L), u(R)) = 0$. This general problem is solved, if possible, using Newton's method which requires an initial guess of the solution. One method for providing an initial guess is by solving the linearized problem $u' = \left(\frac{\partial f}{\partial u}\Big|_{u=u_0}\right) \cdot u$ where u_0 should be either $u(L)$ or $u(R)$. Moreover, it is ideal if the initial function satisfies the boundary conditions.

For example, given interval $\Omega = [0, 1]$. if one of the equations is $u'' = sin(u)$ with boundary condition $u(0) = 1$ and $u(1) = 0$. Set up the problem as a system of first order ODEs: $u' = v$ and $v' = sin(u)$, with the same boundary conditions. Then, solve instead $u' = v \wedge v' = (\frac{d}{du} sin(u)\Big|_{u=1})u = cos(1)u$. The linearized solution is then $u(x) = b(e^{-a(x-2)} - e^{ax}) \wedge v(x) = -ba(e^{a(x-2)} + e^{ax})$ where $a = \sqrt{\cos 1}, b = e^{2\sqrt{a}} - 1$.

The `bvp` class:

```
class bvp {
    public:
    unsigned int max_iterations; // maximum iterations for Newton's method
    double tol; // stopping criteria
    int num_iterations(); // returns the number of iterations needed by the solver
};
```

Variable Order Method

We first introduce a method finite differencing where the order of the polynomial interpolation from the difference formula is derived may be selected by the user. For uniformly spaced data, the method should be $\mathcal{O}(h^{\text{order}})$ where h is the spacing. It is required that `order > 1`.

```
class bvp_k : public bvp
```

We initialize the solver:

```
bvp_k::bvp_k(unsigned int order = 4, double tolerance = 1e-5);
```

where `order` is the order of the interpolating polynomial - 1, and `tolerance` initializes `tol`.

We solve a boundary value problem with:

```
void bvp_k::ode_solve(
    arma::vec& x,
    arma::mat& U,
    const std::function<arma::rowvec(double,const arma::rowvec&)>& odefun,
    const std::function<arma::vec(const arma::rowvec&, const arma::rowvec&)>& bc
    );

void bvp_k::ode_solve(
    arma::vec& x,
    arma::mat& U,
    const std::function<arma::rowvec(double,const arma::rowvec&)>& odefun,
    const std::function<arma::mat(double, const arma::rowvec&)>& jacobian,
    const std::function<arma::vec(const arma::rowvec&, const arma::rowvec&)>& bc
    );
```

We are solving $u' = f(x, u) \wedge g(u(L), u(R)) = 0$, so `odefun` corresponds to f, and `bc` corresponds to g. The function `jacobian` corresponds to the derivative of f with respect to u, i.e. $\frac{\partial f}{\partial u}$. It is necessary that `x` is initialized to a grid of points (of length n) with `x[0]`$= L$ and `x[n-1]`$= R$. The grid `x` should be sorted. The solver will solve the BVP at these grid points, so make sure this grid is dense. It is recommended that `U` is initialized to a guess of the initial problem (purhaps as outlined above), but this is not necessary. Regardless, the matrix must be initialized, this may achieved simply by setting `U = zeros(n,dim)` where `dim` is the dimension of the system of ODEs. The solution will overwrite `U`.

Chebyshev Spectral Method

This method uses a spectrally converging method, where the BVP is solved at Chebyshev nodes scaled to the interval. Because the points are specific, an initial guess must be provided as a function that may be evaluated.

```
class bvp_cheb : public bvp
```

We initialize the solver:

```
bvp_cheb::bvp_cheb(unsigned int num_points = 32, double tolerance = 1e-5);
```

Where `num_points` is the number of points to use in the approximation, and `tolerance` initializes `tol`. Note that for problems with analytic solutions, `bvp_cheb` converges spectrally, thus only few points are ever needed (e.g. <50 points, while `bvp_k` may require >1000 points for a sufficiently accurate approximation).

We solve a boundary value problem with:

```
void ode_solve(
    arma::vec& x,
    arma::mat& U,
    const std::function<arma::rowvec(double,const arma::rowvec&)>& odefun,
    const std::function<arma::vec(const arma::rowvec&, const arma::rowvec&)>& bc,
    const std::function<arma::rowvec(double)>& guess
    );

void ode_solve(
    arma::vec& x,
    arma::mat& U,
    const std::function<arma::rowvec(double,const arma::rowvec&)>& odefun,
    const std::function<arma::mat(double, const arma::rowvec&)>& jacobian,
    const std::function<arma::vec(const arma::rowvec&, const arma::rowvec&)>& bc,
    const std::function<arma::rowvec(double)>& guess
    );
```

Where `odefun`, `jacobian`, and `bc` are just as in `bvp_k`. With `bvp_cheb`, the parameter `x` should be initialized to a vector of length 2 and represent the boundaries of the interval, i.e. `x = {L,R}`. The matrix `U` may be empty. The parameter `guess` is an initial guess of $u(x)$, but may just return a row of zeros you have no guess. It is important that `guess(x).n_elem = dim` for all `x`∈ $[L, R]$, where `dim` is the dimension of the system of ODEs. Both `x` and `U` will be set to the Chebyshev grid and approximate solution. This solution may be interpolated using a polynomial to get an approximation of the solution everywhere on the interval.

Lobatto IIIa method

This method uses a 4th order Lobatto IIIa collocation formula.

```
class bvpIIIa : public bvp
```

We initialize the solver:

```
bvpIIIa::bvpIIIa(double tolerance = 1e-5);
```

where `tolerance` initializes `tol`.

We solve a boundary value problem with:

```
void bvp_k::ode_solve(
    arma::vec& x,
    arma::mat& U,
    const std::function<arma::rowvec(double,const arma::rowvec&)>& odefun,
    const std::function<arma::vec(const arma::rowvec&, const arma::rowvec&)>& bc
    );

void bvp_k::ode_solve(
    arma::vec& x,
    arma::mat& U,
    const std::function<arma::rowvec(double,const arma::rowvec&)>& odefun,
    const std::function<arma::mat(double, const arma::rowvec&)>& jacobian,
    const std::function<arma::vec(const arma::rowvec&, const arma::rowvec&)>& bc
    );
```

Which is treated exactly in the same way as `bvp_k`. This method may be slower than `bvp_k` but is typically more stable (preferable for stiff problems).

Poisson Solver

Given a rectangular region Ω in the x, y plane, we can numerically solve the Poisson/Helmholtz equation $(\nabla^2 + k^2)u = f(x, y)$ with boundary conditions $u(x, y) = g(x, y)$ on $\partial\Omega$ using similar procedures to solving linear ODEs.

We solve the problem with the function:

```
void poisson_helmholtz_2d(
    arma::mat& X,
    arma::mat& Y,
    arma::mat& U,
    const std::function<arma::mat(const arma::mat&, const arma::mat&)>& f,
    const std::function<arma::mat(const arma::mat&, const arma::mat&)>& bc,
    double eig = 0,
    int num_grid_points = 32);
```

We initialize `X` with the bounds in x, i.e. `X = {xLB, xUB}`, and the same for `Y`, i.e. `Y = {yLB, yUB}`. The matrix `U` does not need to be intialized. The function `bc` should equal `g(x,y)`. The parameter `eig` is k, which is `0` by default corresponding to Poisson's equation.

`X,Y,U` will be overwritten with the grid points solved for.

The parameter `num_grid_points` is the number of grid points along each axis, meaning the total number of points solved for will be `num_grid_points`ˆ2. This solver uses the Chebyshev spectral order method only. The solver is take $\mathcal{O}(n^6)$ with respect to `num_grid_points`.